Midterm Exam

```
    ____ ____ ___ ____    _______ __
   / ____/ ____/ ___/ ____/  /___ /___  / _ \
  /  _/ /  _// /    \___ \    /_ \  / / / / /
 / /___/ /__/ /___ ___) /  ___) / / / /_/ /
/_____/_____\____/____/  /____//_/  \___/
```

EECS 370 Fall 2022: Introduction to Computer Organization

You are to abide by the University of Michigan College of Engineering Honor Code. Please sign below to signify that you have kept the honor code pledge:

I have neither given nor received aid on this exam,
nor have I concealed any violations of the Honor Code.

Signature: ______________________________

Name: *ANSWER KEY* ______________________________

Uniqname: ______________________________

First/Last name of person sitting to your ***Right***
(Write ⊥ if you are at the end of the row) ______________________________

First/Last name of person sitting to your ***Left***
(Write ⊥ if you are at the end of the row) ______________________________

Exam Directions:

- You have **120 minutes** to complete the exam. There are **9** questions in the exam on **22** pages (double-sided). **Please flip through your exam to ensure you have all 22 pages.**
- You must show your work to be eligible for partial credit!
- Write legibly and dark enough for the scanners to read your answers.
- **Write your uniqname on the line provided at the top of each page.**

Exam Materials:

- You are allotted **one 8.5 x 11 double-sided** note sheet to bring into the exam room.
- You are allowed to use calculators that do not have an internet connection. All other electronic devices, such as cell phones or anything or calculators with an internet connection, are strictly forbidden.

1.	**Short Questions**	________ /	**20 pts**
2.	**Memory Alignment**	________ /	**7pts**
3.	**Caller and Callee**	________ /	**6 pts**
4.	**Linker**	________ /	**6 pts**
5.	**C to ARM**	________ /	**15 pts**
6.	**SC and MC Performance**	________ /	**6 pts**
7.	**New ISA**	________ /	**10 pts**
8.	**LC2K Single-Cycle Design**	________ /	**10 pts**
9.	**LC2K Multi-Cycle Design**	________ /	**20 pts**
	TOTAL	________ /	**100 pts**

1. Short Questions [20 pts]

Complete the following true/false and short answer questions.

True/False Questions — Circle One:

a) **[1 pt]** IEEE floating point uses 2s complement to represent negative values F

b) **[1 pt]** It is possible to compile any C program to ARM or x86. T

c) **[1 pt]** Backwards compatibility means any new instruction added to an ISA must benefit programs written in the past F

d) **[1 pt]** Input to a program (through command line and I/O) does not impact the execution time of that program running on a single-cycle datapath processor. F

e) **[1 pt]** The CPI of a multicycle processor is sometimes less than that of a single cycle processor F.

f) **[1 pt]** Circle all memory regions whose sizes are known at compile time.

i) [✓ Text (instruction segment)] ii) [✓ Static data segment]

iii) Heap iv) Stack

g) **[1 pt]** Circle all instructions that use relative addressing mode in LC2K.

i) load (lw) ii) store (sw)

iii) [✓ beq] iv) jalr

8+1=9

8 4 2 1

1 0 1
4 2 1

h) [2 pts] What is the machine code for **`lw 2 5 64`** in LC2K in hexadecimal?

opcode A B

`Opcode for lw is 0b010`

0 | 10 | 01 | 0 | 101 0000 0000 0100 0000

128 64 32 16 8 4 2 1

15 - 0.

`I-type instruction`
`(lw)`
0
	`bits 24-22: opcode`
	0 9 5 0 0 4 0
	`bits 21-19: reg A`
	`bits 18-16: reg B`
	`bits 15-0: offsetField (a 16-bit, 2's complement number with a range of -32768 to 32767)`

Answer:

0x950040.

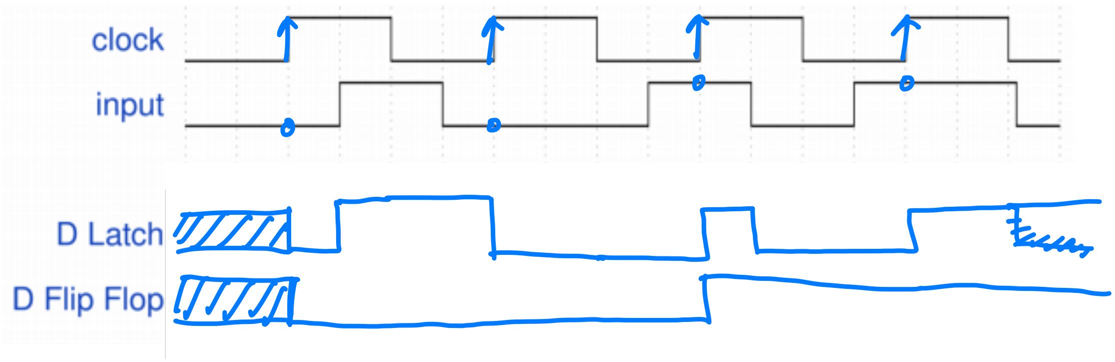
i) [4 pts] Write the value 0xFBC4 to memory address 1000.

Byte-addressable, Little Endian	
Address	**Value**
1000	C4
1001	FB

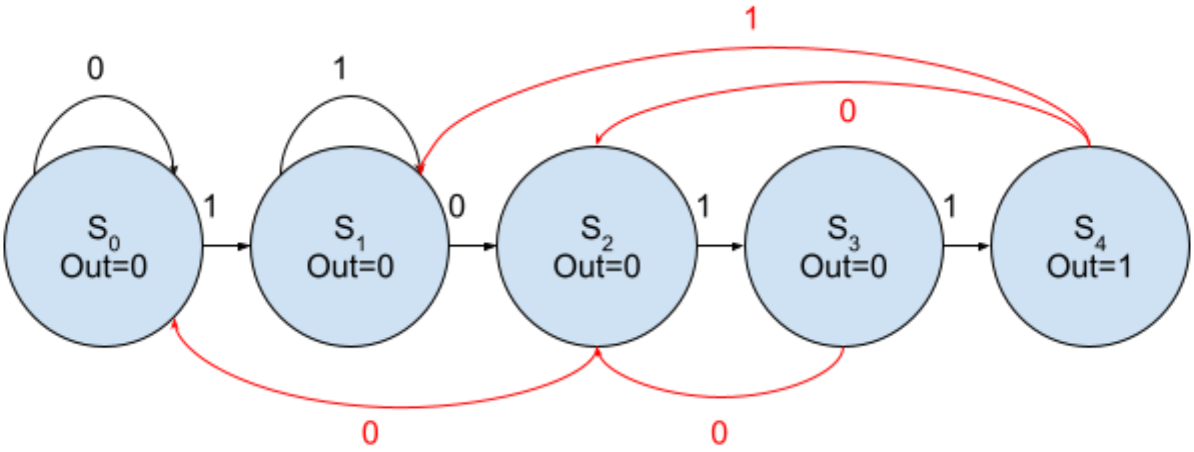
Byte-addressable, Big Endian	
Address	**Value**
1000	FB.
~~999~~ 1001	C4

2-byte addressable, Little Endian.	
Address	**Value**
1000	FBC4.
1001	

j) **[3 pts]** Draw the wave-forms for D-latch and positive edge-triggered D-flip flop on the timing diagram below. Clock serves as a gating input for the D latch.



k) **[4 pts]** Complete the following FSM to detect a 4-digit binary sequence “1011”. Input to the FSM is a string of binary digits. FSM needs 4 **arrows** (with input label) or state transitions to be complete.



2. Memory Alignment [7 pts]

Assume **64-bit**, byte-addressable architecture, and that the compiler ensures aligned memory accesses by using padding. “outer” begins at address 1000 (in decimal).

```
typedef struct {          1012- 1015
      int x; 4            1016 - 1017
      short y; 2
} widget_t; 8             1012-1017 → 6.          8×4=32.
                          1018-1019 → 2 padding.
struct { 8
      int *a;  1000 - 1007
      char b;  1008  < 1009-1011
      widget_t c[4]; 1012 - 1043
      double d;
} outer;    8  1048 < (1044-1047)
               48-1055
1000-1055  56
```

a) **[2 pts]** What is the size of widget_t? 8. bytes

widget_t must be stored at an address divisible by 4.

b) **[5 pts]** Specify the memory layout for “outer” by completing the table below.

Variable Name	Size (bytes)	Start Address (in decimal)	End Address, inclusive (in decimal)
a	8	**1000**	1007
b	1	1008	1008
c	32	1012	1043
d	8	1048	1055
outer	56	1000	1055.

3. Caller and Callee [6 pts]

Complete the caller/callee saved registers for the following C program.

Assume the compiler checks for liveness across function calls. It does not perform any other optimizations.

```
void foo() ←
{
    int cnt = 10;
    int a = 0, b = 0;

    bar();  1
    b = 10;

    for (int i=0; i < 10; i++) {
        a = a + b;
        bar();
        b = i;  10
    }
} ←
```

```
 void bar()
{  ← 1
     int x = 1, y = 2;

     for(int x = 0; x < 10; x++)
     {
       y++;
     }

 } ←



              11
```

Assume foo() is invoked once, and bar() is only invoked by foo().

Complete 2nd and 3rd columns to specify the **dynamic** number of additional load/store **pairs** executed to save and restore registers for each variable, if we use only caller-saver or only callee-save registers.

Next, assume there are only **2 callee** and **2 caller** registers. In the fourth column of the table, allocate a caller or a callee register to a variable such that the number of additional loads/stores executed is minimized.

Variable	Caller Save	Callee Save	Caller or Callee?
`foo: a`	11	1	Callee.
`foo: b`	0	1	Caller
`foo: cnt`	0	1	Caller.
`foo: i`	10	1	Callee.
`bar: x`	0	11	Caller
`bar: y`	0	11	Caller.

4. Linker [6 pts]

Complete the symbol and relocation tables for the following C program.

Fill in the symbol and relocation tables for the following two C files. You may not need to use all the table entries.

main.c

```c
#include <stdlib.h>

int bonus = 9;
extern int  getFizz(int num);

int main() {
  int * fizz = malloc(1024 * 4);
  for(int num = 0; num < 1024; ++num)
  {
    fizz[num] = getFizz(num);
  }
}
```

FizzBuzz.c

```c
extern int bonus;

int getFizz(int num) {
  int tmp = bonus % num;
  bonus = tmp;

  if (num % 3 == 0) {
    num = 3;
  }
  else if (num % 5 == 0) {
    num = 5;
  }

  return num + bonus;
}
```

Types:

T: Text
D: Data
U: Undefined

main.o Symbol Table [2 pts]	
Symbol	Type (T/D/U)
bonus	D
getFizz	U
malloc	U
main	T

FizzBuzz.o Symbol Table [1 pt]	
Symbol	Type (T/D/U)
bonus	U
getFizz	T

Note: The following code is identical to the one on the previous page.

main.c

```
1   #include <stdlib.h>
2
3   int bonus = 9; declare
4   extern int getFizz(int num); 3
5
6   int main() {
7     int * fizz = malloc(1024 * 4);
8     for(int num = 0; num < 1024; ++num)
9     {
10      fizz[num] = getFizz(num);
11    }
12  }
13
14
15
```

FizzBuzz.c

```
1   extern int bonus;
2
3   int getFizz(int num) {
4     int tmp = bonus % num;
5     bonus = tmp;
6
7     if (num % 3 == 0) {
8       num = 3;
9     }
10    else if (num % 5 == 0) {
11      num = 5;
12    }
13
14    return num + bonus;
15  }
```

Instructions:

ldur (load)
stur (store)
bl (branch with link)

main.o Relocation Table [1 pt]		
Line #	Symbol	Instruction
~~3~~	~~bonus~~	~~ldur~~
7	malloc	bl
10	getFizz	bl

FizzBuzz.o Relocation Table [2 pt]		
Line #	Symbol	Instruction
4	bonus	ldur
5	bonus	stur
14	bonus	ldur

总结: relocation table ① 调用 < variable / function.

②※Declare不管 在 symbolic里面.

5. C to ARM [15 pts]

Convert C program to ARM assembly.

Convert the following C code to ARM assembly. The assembly code should be ABI compliant. Each register is 64 bits. The starting address of **radios[]** is mapped to **X1** when it's passed to **whats_new()**. The input argument of **hear()** is mapped to **X1**.

C

```
struct Radio {
    int32_t noise;   // 4B
    int32_t gaga;    // 4B
};

int32_t hear(int32_t sound) {
x1 →sound += 8;
    return sound;
}
                                      x1
void
whats_new(struct Radio radios[]) {
    int64_t i = 0;   ← x5
    for ( ; i < 10; ++i) {   ← x5
        radios[i].gaga++;   x7
        if (i < 5) {
            radios[i].gaga *= 4;   ← x7
        } else {
            radios[i].noise =
                hear(radios[i].gaga);
        }                             x1 = x7
    }
}
```

ARM

return value.

```
hear:
        ADDI    ~~X1~~ X0,  X1,  #8
return:
        BR      X30
whats_new:
        MOV     X2,  X1
        MOVZ    X5,  #0
loop:                                   x6 = x5 _ 3
        CMPI    X5,  #10
        B.EQ    end       X6 = X5
        LSL     X6,  X5,  #3
        ADD     X6,  X6,  ~~X~~ X2
        LUDRSW  X7,  [X6, #4]
        ADDI    X7,  X7,  #1
        STURW   X7,  [X6, #4]
        CMPI    X5,  #5
        B.GE    else
if:
        LSL     X7,  X7,  #2
        STURW   X7,  [X6, #4]
        B       inc
else:
        MOV     X1,  X7
        BL      hear
 return STURW   X0,  [X6, ~~#4~~] #0.
inc:
        ADDI    X5,  X5,  #1
        B       loop
end:
```

6.	**SC and MC Performance** **[6 pts]**
	Analyze the performance of LC2K Single-Cycle and Multi-Cycle datapaths.

Consider the LC2K single-cycle and multi-cycle data paths from lecture. The components in the datapath has the following delays:

170 from10

lw : 60 + 5 + 60 + 15 + 10 ✓

`Read register file:`	`5 ns`
`Write register file:`	`10 ns`
`ALU:`	`15 ns`
`Read memory:`	`60 ns`
`Write memory:`	`20 ns`
`All other components:`	`0 ns`

✓

read regiseer

ALU ✓

reg = mem[reg + offset]

↑

Write register file.

Read memory ✓

(a) Calculate the minimum **clock period in ns** for following designs:

Single-Cycle [2 pts]	**Multi-Cycle [1 pt]**
150ns	60ns

(b) Consider a program that executes 100 instructions with the following mix:

20%	sw	20	(4)
30%	add/nor	30	(4)
20%	lw	20	(5)
20%	beq	20	(4)
10%	noop	8	(3).

20×4 + 30×4 + 20×5 + 20×4

80 + 120 + 100 + 80

200 + 180 = 380

380 + 20 = 400 400×60

=2400

What is the total program runtime (in nanoseconds) for both single-cycle and multi-cycle data paths?

Single-Cycle [1 pt]	**Multi-Cycle [2 pts]**
15000 ns	~~420 ns.~~ 2400.

7.	**New ISA** **[10 pts]**
	Program using a new ISA (extended from LC2K)

Consider a new ISA called BC3K (Big Computer 3000). BC3K is identical to LC2K except for the following differences:

- BC3K has **16 registers**. Each register is **32-bits** (same as LC2K)
- 4 new instructions have been added:

Instruction	**Description**
ext_add rA, rB, rC, rD	{rC, rD} = rA + rB Adds the unsigned 32-bit registers rA and rB to produce an **unsigned 64-bit result**. Bits 63-32 of the results are stored in rC Bits 31-0 of the result are stored in rD
ext_mul rA, rB, rC, rD	$\frac{1}{2}\,\frac{1}{1}\quad\frac{1}{2}\,\frac{1}{1}\quad\boxed{\frac{1}{8}\,\frac{0}{4}}\,\frac{0}{2}\,\frac{1}{1}$ {rC, rD} = rA * rB Multiplies the unsigned 32-bit registers rA and rB to produce an **unsigned 64-bit result**. $3*3=9$ Bits 63-32 of the results are stored in rC Bits 31-0 of the result are stored in rD
cmov rA, rB, rC	If ([rA] != 0) [rC] = [rB] else do nothing

cmp rA, rB, rC	if ([rA] == [rB]) [rC] = 1
	else [rC] = 0

Note: Although ext_add and ext_mul produce 64-bit results, their inputs are only 32-bits.

(a) [3 pts] Fill in the blanks to translate the C code to BC3K. Use the following register mappings:

Variable	Register
a	1
b	2
out	15
<temporary value>	3

C code	BC3K code

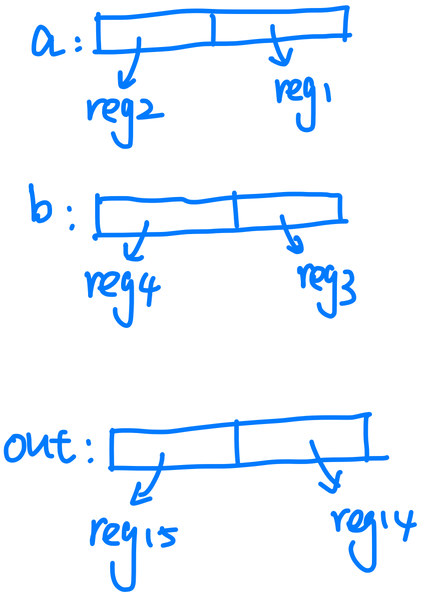
`int a;`
`int b;`
`int out;`
 	
`if(a != b) {`
`  15out = a + b;`
`       1     2`
`} else {`
`    out = a;`
`}`
	cmp 1 2 3
	
	add 1 2 15
	
	cmov 3 1 15

(b) [3 pts] Emulate 64-bit add.

Complete the following BC3K assembly code to emulate a 64-bit add operation in the C code shown below. Any bits in the resulting sum which overflow beyond 64 bits may be ignored.

Use the following register mappings:

Variable	Register
a, bits 31-0	1
a, bits 63-32	2
b, bits 31-0	3
b, bits 63-32	4
out, bits 31-0	14
out, bits 63-32	15



uint64_t type in C refers to 64-bit unsigned value.

C code	BC3K code

uint64_t a;
uint64_t b;
uint64_t out;
 	
out = a + b;
	out
	ext-add 1 3 15 14
	add 2 15 15
	a
	add. 4 15 15

b

(c) [4 pts] Fill in the blanks to translate the code description to BC3K.

Use the following register mappings:

Variable	Register
a	1
b	2
<temporary value>	3
~~overflow~~	~~4~~
out	5
<temporary value>	6

$1\ \frac{0}{2} \times \frac{1}{2}\ 0 = \frac{0}{8}\boxed{\frac{1}{4}}\frac{0}{2}\frac{0}{1}$

$\underset{3}{\underline{1\ 1}} \times \underset{3}{\underline{1\ 1}} = \frac{1}{8}\frac{0}{4}\frac{0}{2}\frac{1}{1}$

reg3 :#1

reg4 =0

uint32_t type in C refers to a 32-bit unsigned value.

Code description	BC3K code

uint32_t a;
uint32_t b;
uint32_t overflow;
uint32_t out;
 	
Calculate the product **a** * **b** and store the 32-bit result in **out**. If the product does not fit in 32 bits, set **overflow** to 1. Otherwise, set **overflow** to 0.
	lw 0 3 one
	add 0 0 4 ←overflow
	ext_mul 1 2 6 5
	↑
	cmov ~~cmp~~ 6 ~~4~~ ? ~~6~~
	3 4 ?
	halt
	overflow存在4
	one .fill 1

8. LC2K Single-Cycle Design [10 pts]

Extend the LC2K Single-Cycle Datapath to compute new instructions!

Add a new I-type instruction *Jump and Store (jas)* to LC2K:

jas regA regB immediate

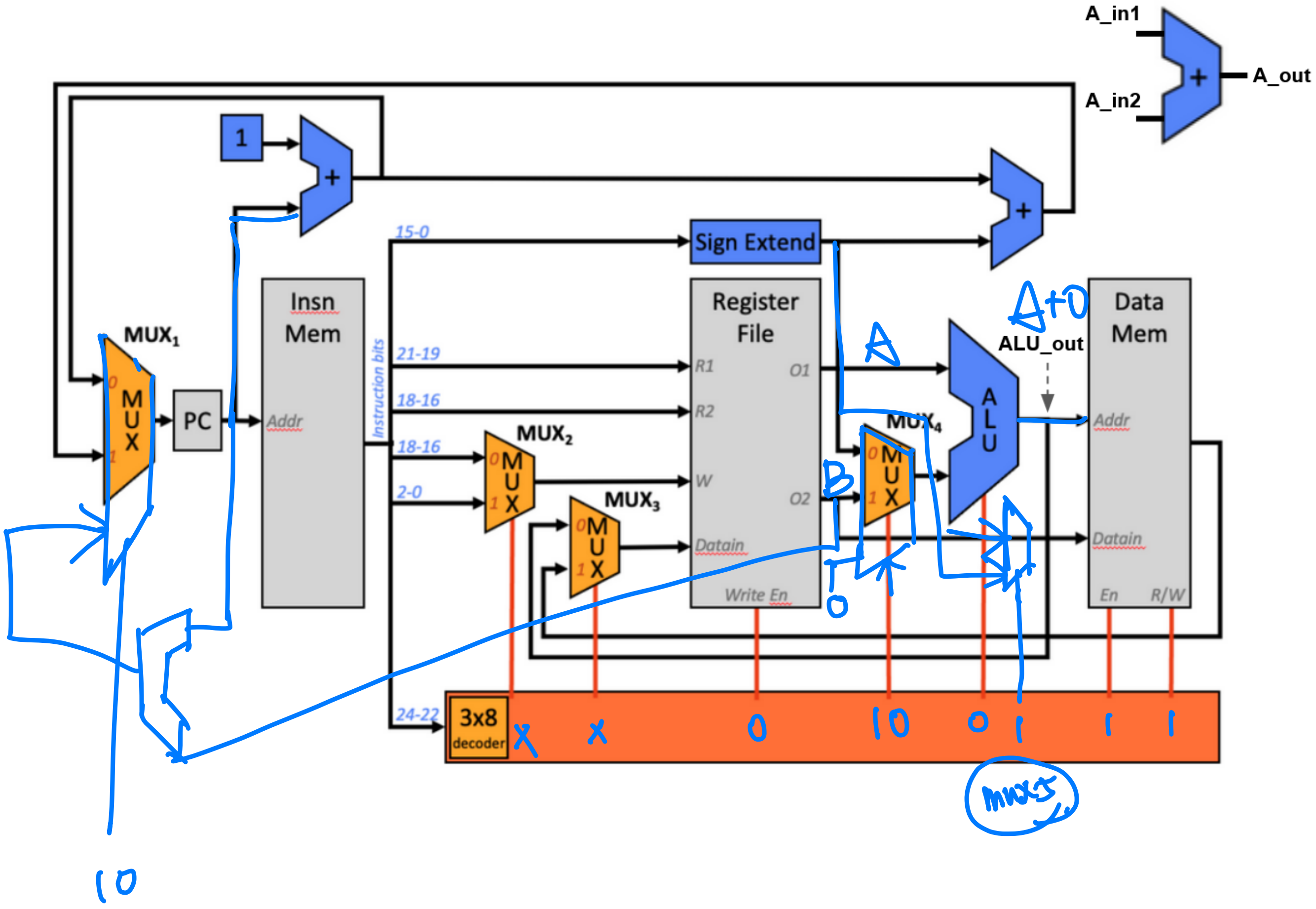
Its semantics are:

$$mem\,[[regA]] \;=\; sign\,extend(immediate)$$
$$PC \;=\; PC \;+\; [regB]$$

[regA] is the value in regA.
mem[[regA]] is the value stored at memory address [regA].

Modify the LC2K single cycle datapath from lecture shown below to support *jas* (note: picture shown doesn't support *jalr*). You are restricted to the following modifications:

- **One** new adder (shown in the top-right corner of the picture below)
- May add **one** new 2:1 MUX (MUX5)
- Extend up to **two** old 2:1 MUXes (add at most two inputs to each extended MUX)
- Add zeroReg that holds a constant 0 value



a) **Describe your modifications answering the following questions.** **[6 pts]**

i) Specify the extensions made to old MUXes (MUX1, MUX2, MUX3, or MUX4).

Specify the MUX number (1-4), and new inputs added (can be either one or two inputs). Use index 10 if only one new input is added.

MUX#	New input (index 10)	New input (index 11)
MUX_1 ✓	A_out. ~~Pc + [reg~~ B]	
MUX_4 ✓	0 ✓	

ii) Specify the inputs and output for the new adder.

Additional Adder	Wire Connected
A_in1	O_2 ✓ [regB].
A_in2	PC ✓
A_out	MUX_1 (index 10) ✓

iii) Specify the input and output for the new MUX5.

MUX5	Wire Connected
Input (0)	offset ✓
Input (1)	O_2 ✓ [regB]
Output	Data in (of memory) ✓

b) Determine the control signals for *jas*. **[4 pts]**

Control bit of ALU: 0 add; 1 nor

Control bit of Data memory: 0 read; 1 write

MUX1	PC_en	MUX2	MUX3	Reg en	MUX4	ALU	Data mem en	Data mem R/W	MUX5
10 ✓	1 ✓	X ✓	X ✓	0 ✓	10 ✓	0 ✓	1 ✓	1 ✓	1 ✓

9. LC2K Multi-Cycle Design [20 pts]

Extend the LC2K Multi-Cycle Datapath to compute new instructions!

Consider the following new LC2K I-type conditional store instruction.

SWEQ	**S**tore **W**ord on **Eq**uality.
Reg_A Reg_B $Immediate$	*if [Reg$_B$] == mem[[Reg$_A$]]* $mem[\,[RegA]\,] \; = sign\,extend(\,imm)$

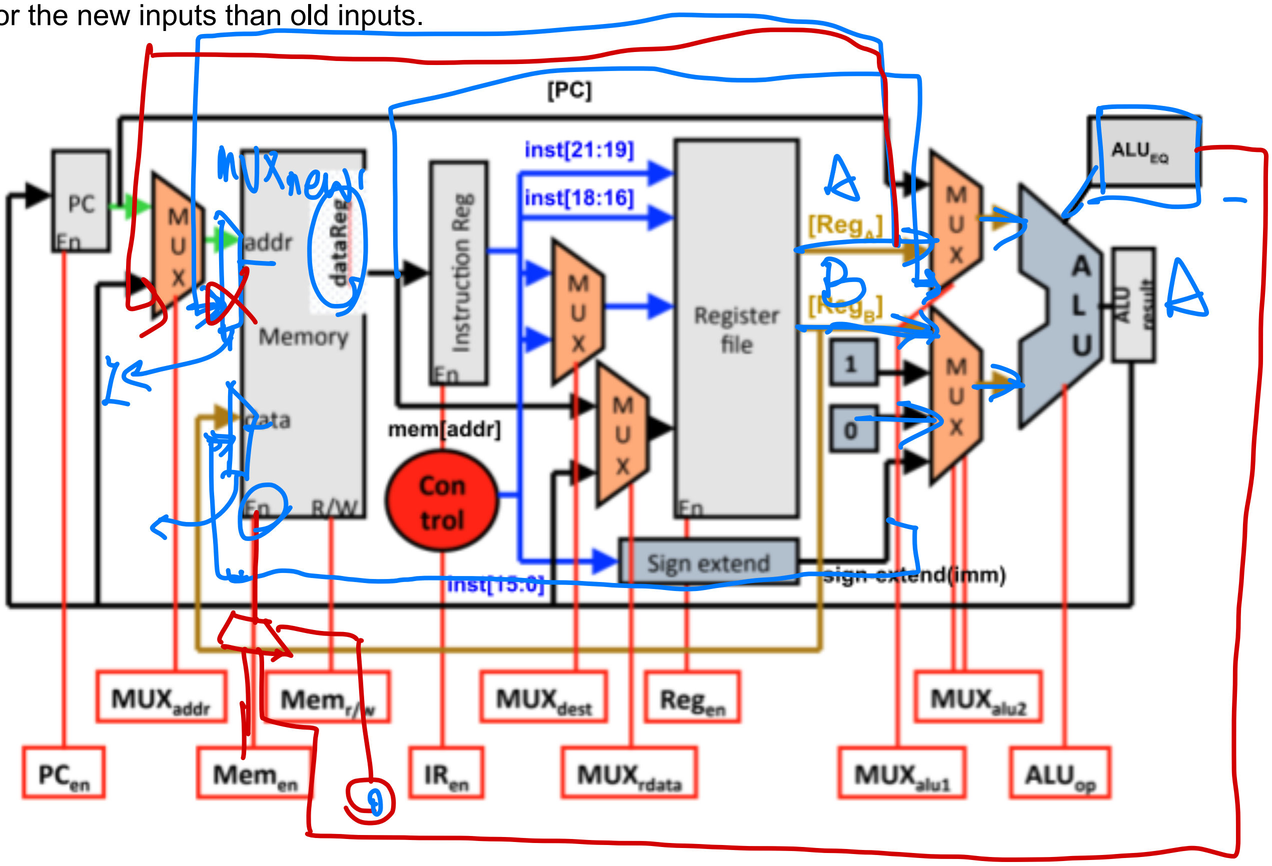
[RegA] is the value in RegA.
mem[[regA]] is the value stored at memory address [regA].

Modify the LC2K multi-cycle datapath to support SWEQ. You have the following restrictions:

- **Modify up to two MUXes** by adding at most one additional input
- **Add up to two new MUXes** (MUXnew1 and MUXnew2).

Note: Instruction-Reg, dataReg, ALUeq, and ALUresult are temporary registers.

Assume new inputs are added at the "bottom" of the MUX as the last input. That is, select control is higher for the new inputs than old inputs.



Implement **SWEQ** in **five** cycles. Fetch and decode cycles remain the same as other instructions.

a. What operations take place in cycles 3, 4 and 5? Follow the format specified below.

Use "**dataReg**" to specify the temporary register into which the value read from memory is written to (similar to how **lw** reads from memory in its ***4th cycle***). ***[6 pts]***

Cycle 1:

$[IR] = Mem[PC]$

$[ALU_{Result}] = [PC] + 1$

Cycle 2:

$[PC] = ALU_{Result}$

$Decode\ instruction;$ *Read registers*

Cycle 3:

dataReg = mem[[regA]].

Cycle 4:

ALU_{eq} = [mem[[regA]] == [regB]

Cycle 5:

mem[[regA]] = sign extended.

b. Specify old MUX(es) (MUX_{addr} MUX_{dest} MUX_{rdata} $MUX_{alu1,}$ or MUX_{alu2}) that need an additional input, and what those input wires are. ***[4 pts]***

If you choose not to add additional input(s), please indicate this by putting N/A in both rows.

MUX with additional input	***New wire connected***
MUXalu1	dataReg
MUXaddr	[regA]

c. What are the inputs and output wires for ***new MUXes?*** If you choose ***not*** to add a ***new MUX***, please indicate this by answering ***N/A.*** ***[4 pts]***

Option 1:

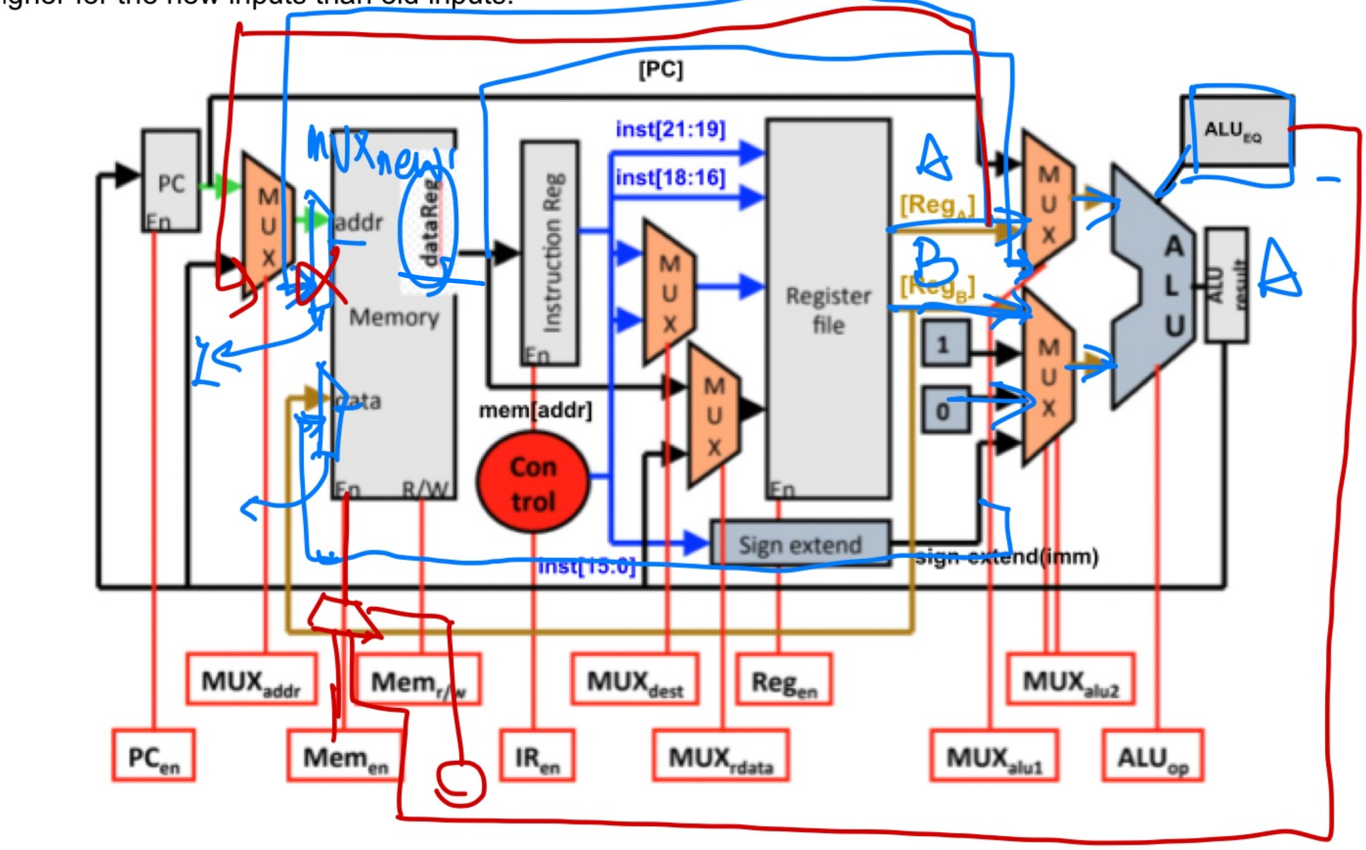
MUX	***input 0***	***input 1***	***output***
MUXnew1	[regB]	signextend.	data.
MUXnew2	Memen	ALU_{EQ}.	En.

d. Specify the control signals for each cycle.
The control signal for newly added MUX input (from part **b**) is **10**.
The control signal to the new MUXes (part **c**) can be **0** or **1** or **X**. Leave MUXnew1 and/or MUXnew2 columns empty if they were not used. ***[6 pts]***

Option 1:

Cycle	PC en	MUX addr	Mem en	Mem r/w	IR en	MUX dest	MUX rdata	Reg en	MUX alu1	MUX alu2	ALU op	MUX new1	MUX new2
1	0	0	1	0	1	X	X	0	0	01	0	~~X~~ 0	X
2	1	X	0	X	0	X	X	0	X	X	X	~~X~~ 0	X
3	0	10	1	0	0	X	X	0	X	X	X	~~X~~ 0	X
4	0	X	0	X	0	X	X	0	10	0	X	~~X~~ 0	X
5	0	10	X	1	0	X	X	0	X	X	X	1	1

ssume new inputs are added at the "bottom" of the MUX as the last input. That is, select control is igher for the new inputs than old inputs.



All possible answers

Option 1:

Cycle	PC en	MUX addr	Mem en	Mem r/w	IR en	MUX dest	MUX rdata	Reg en	MUX alu1	MUX alu2	ALU op	MUX new1	MUX new2
1	0	0	1	0	1	X	X	0	0	01	0	0	X
2	1	X	0	X	0	X	X	0	X	X	X	0	X
3	0	10	1	0	0	X	X	0	X	X	X	0	X
4	0	X	0	X	0	X	X	0	10	00	X	0	X
5	0	10	X	1	0	X	X	0	X	X	X	1	1 / 0

Cycle	PC en	MUX addr	Mem en	Mem r/w	IR en	MUX dest	MUX rdata	Reg en	MUX alu1	MUX alu2	ALU op	MUX new1	MUX new2
1	0	0	1	0	1	X	X	0	0	01	0	X	0
2	1	X	0	X	0	X	X	0	X	X	X	X	0
3	0	10	1	0	0	X	X	0	X	X	X	X	0
4	0	X	0	X	0	X	X	0	10	00	X	X	0
5	0	10	X	1	0	X	X	0	X	X	X	1 / 0	1

Cycle	PC en	MUX addr	Mem en	Mem r/w	IR en	MUX dest	MUX rdata	Reg en	MUX alu1	MUX alu2	ALU op	MUX new1	MUX new2
1	0	0	1	0	1	X	X	0	0	01	0	1	X
2	1	X	0	X	0	X	X	0	X	X	X	1	X
3	0	10	1	0	0	X	X	0	X	X	X	1	X
4	0	X	0	X	0	X	X	0	10	00	X	1	X
5	0	10	X	1	0	X	X	0	X	X	X	0	1 / 0

Cycle	PC en	MUX addr	Mem en	Mem r/w	IR en	MUX dest	MUX rdata	Reg en	MUX alu1	MUX alu2	ALU op	MUX new1	MUX new2
1	0	0	1	0	1	X	X	0	0	01	0	X	1
2	1	X	0	X	0	X	X	0	X	X	X	X	1
3	0	10	1	0	0	X	X	0	X	X	X	X	1
4	0	X	0	X	0	X	X	0	10	00	X	X	1
5	0	10	X	1	0	X	X	0	X	X	X	1 / 0	0

Option 2:

Cycle	PC en	MUX addr	Mem en	Mem r/w	IR en	MUX dest	MUX rdata	Reg en	MUX alu1	MUX alu2	ALU op	MUX new1	MUX new2
1	0	0	1	0	1	X	X	0	0	01	0	X	X
2	1	X	0	X	0	X	X	0	X	X	X	X	X
3	0	10	1	0	0	X	X	0	X	X	X	X	X
4	0	X	0	X	0	X	X	0	10	00	X	X	X
5	0	10	1	1	0	X	X	0	X	X	X	ALU EQ	1 / 0

Cycle	PC en	MUX addr	Mem en	Mem r/w	IR en	MUX dest	MUX rdata	Reg en	MUX alu1	MUX alu2	ALU op	MUX new1	MUX new2
1	0	0	1	0	1	X	X	0	0	01	0	X	X
2	1	X	0	X	0	X	X	0	X	X	X	X	X
3	0	10	1	0	0	X	X	0	X	X	X	X	X
4	0	X	0	X	0	X	X	0	10	00	X	X	X
5	0	10	1	1	0	X	X	0	X	X	X	1 / 0	ALU EQ